

---

# INNOVATIONS IN CODE MAINTENANCE

**BY DR TOBY SUCHAROV OF ERUDINE**

**A WHITE PAPER FROM ERUDINE**

---

<b>3</b>	<b>EXECUTIVE SUMMARY</b>
<b>4</b>	<b>INTRODUCTION / RESEARCH</b>
<b>5-7</b>	<b>CODE MAINTENANCE – TRADITIONAL SOLUTIONS</b> <b>REFINEMENT / CHANGE / NEW FUNCTIONALITY – ADDING NEW CODE / REFACTORING</b>
<b>8-10</b>	<b>WHAT ARE THE ALTERNATIVES?</b> <b>CODE MAINTENANCE: TAKING A LESSON FROM COCOMO</b>
<b>11-12</b>	<b>SOLUTIONS TO CODE MAINTENANCE</b> <b>ERUDINE – A MAINTENANCE REVOLUTION / REFINEMENT AND ERUDINE / CHANGE AND ERUDINE / ADDING NEW FUNCTIONALITY AND ERUDINE / REFACTORING</b>
<b>13</b>	<b>ERUDINE ADDRESSES THE LESSONS OF COCOMO</b> <b>INFLEXIBILITY TO NATURAL ADAPTATION / UNPRECEDENTED PROJECTS / WEAK COMMUNICATIONS / PRODUCT FACTORS / WEAK DEVELOPMENT TOOLS / PLATFORM TUNING / PERSONNEL FACTORS</b>
<b>15</b>	<b>THE FUTURE OF CODE MAINTENANCE / BIBLIOGRAPHY</b>

## EXECUTIVE SUMMARY

---

**Within IT, code maintenance is an often overlooked area of misery and costs. Code maintenance is generally overlooked as the costs and potential problems associated with cleaning up old code can be huge. The industry sees their options are to leave the code as is and accept the massive code maintenance costs or to open a pandoras box by attempting to fix the code. Neither option is desirable, leaving the code alone and maintaining it only when necessary often seems like the lesser of two very significant evils. According to many research papers, the industry seems to accept that the cost of maintaining code will always be high.**

This paper takes a close look at traditional methods of maintaining code, specifically the key maintenance issues:

- Refinement.
- Change.
- New functionality.
- Refactoring.

To illustrate key points of the difficulties of code maintenance we consider COCOMO 2 – the COConstructive COst MOdel 2. This is a model that estimates the cost and time for building projects based on the attributes of the project. The paper then considers alternatives to traditional maintenance methods. Finally we offer the industry a real solution to the problems and costs of successfully maintaining code. The solution is the Erudine Behaviour Engine, this interweaves traditional methods with new cutting edge technology. This white paper is one of a number of papers that show how the Erudine Behaviour Engine is positioned within the industry. It is drawn from the foundation-stone white paper – Unlocking Tacit Knowledge by Conclusion and Justification. This paper can be viewed and downloaded from [www.erudine.com](http://www.erudine.com)

## INTRODUCTION

---

Considering for a moment some of the major IT projects that have automated complex business processes in many industries, we can recognise the huge benefits that these successful projects have brought to our lives. As developers we know from literature, studies and our own experience that during code maintenance about fifty-percent of a developer's time can be spent in getting to understand the code. Successful projects are therefore expensive in terms of the time and money needed to maintain their code base. Obviously, code maintenance should be a critical issue for the global IT industry, but the question this paper poses is – 'Have the benefits of successful IT projects blinded us to the continually high costs of maintaining code?' Let's consider the plethora of research conducted on code maintenance.

## RESEARCH

---

**Research shows clear evidence that the problem of code maintenance is a massive and critical issue and it is getting worse, not better.**

Sutherland (1995) stated that the annual cost of code maintenance in the USA has been estimated at around \$70 billion dollars. The most famous, or infamous, example of code maintenance was in the year 2000 when millennium-bug related problems cost businesses and government departments in the US a staggering one-hundred billion dollars.

Eastwood (1993) reports that the proportion of spending on code maintenance was 75% of the total IT expenditure in Fortune 1000 companies. Erlikh (2000) stated that code maintenance and evolution costs take up more than 90% of total code cost.

Ulrich (1990) reported that approximately 120 billion lines of source code were being actively maintained. Sommerville (2000) states that this figure has now grown to 250 billion lines of source code.

Examples of the cost and pain of code maintenance can be found everywhere within IT.

The issue of code maintenance leads us to ask a number of questions:

- What are the root problems specific to code maintenance and are these problems unsolvable?
- Can the industry improve the process by which code is maintained?
- What innovations are there that will impact code maintenance in the future?

## CODE MAINTENANCE – TRADITIONAL SOLUTIONS

---

**Traditional solutions to the vast cost of code maintenance always start with, and focus heavily upon, the personnel aspect of a project. No article on code maintenance can be taken seriously without the obligatory discussion on these.**

It is generally the case that code maintenance is conducted by second-tier programmers, leaving the first-tier programmers to green field projects. When we consider that first-tier developers are put on projects that have a projected 30% rate of success (see our white paper *Unlocking Tacit Knowledge by Conclusion and Justification* for more details) and that second-tier developers are set to work on code maintenance for mission critical profitable projects, we have to wonder about the wisdom of such allocations.

While it is true that code maintenance problems would be reduced by allocating more experienced developers to these problems the sad fact is that there are not enough experienced developers to spread between new and old projects. We will now look at the more fundamental project issues that affect code maintenance.

It has been observed that there are four main types of maintenance:

1. Refinement.
2. Change.
3. New functionality.
4. Refactoring.

**REFINEMENT**

Refinement was dealt with in greater detail in the whitepaper *Unlocking Tacit Knowledge by Conclusion and Justification* available from [www.erudine.com](http://www.erudine.com)

The essential point here is that for a complex project, the expert is unable to state all the requirements in advance: some only become apparent when the need for the requirement occurs. In most instances refinement requires the code base or the business behaviour to be changed. At this point an IT developer normally approaches changing the code with a sense of trepidation.

With a complex or large system it is almost inevitable that refinements will generate side effects that cannot be predicted. The difficulty of refinement can be clearly seen by examining the process that a developer typically adopts when refining code:

**Refinement stages**

Operational stages	Necessary activity
Estimate/Negotiate the cost of change.	
Update the requirements specification.	The new requirements need to be made explicit.
Track changes through a traceability matrix.	All the impacted artefacts need to be full identified, for example, design documents, code, drawings and tests.
Change the design documents.	The design documents need to be updated to show the new changes.
Change the code.	Before the code can be changed, the developers need to spend time understanding the code. The code change itself may be simple – adding IF/THEN to deal with a corner case – or profound, requiring architectural change. It is often the case that it is only at this point that the true scope of the change becomes apparent.
Change the tests.	Some of the test suites impacted by the change will be identified by the traceability matrix, but significantly others will only be identified when they fail.
Deploy to testing.	The code base is moved to testing rigs, with change notes and new tests prepared.
Execute the test suite.	Independent testers will test the new suite with formal and ad hoc tests.
Deploy to staging.	The code is moved into the final staging area and then marked as being a new release.
More testing.	Often more tests are performed prior to deployment.
Deploy to live.	The code now goes live.

### Why is this process so complex, time consuming and so expensive?

Changing live code is very dangerous because of the unexpected side effects that can, and invariably do, occur. The nightmare scenario is always that a bug brings the whole system crashing down and any downtime will create huge costs for the company.

These unforeseen bugs are an inevitable consequence of building large and complex projects, the developer cannot maintain a full mental model of the code and is not able to fully predict changes to the code base.

### CHANGE

The need to change the functionality of the code is another inevitable consequence of the evolution of business practices with time. Most code development processes handle this type of maintenance in the exactly the same way that they handle refinement.

### NEW FUNCTIONALITY – ADDING NEW CODE

Adding new code is perhaps the easiest modification to undertake as the developer is free to start work from a blank sheet to create and implement the new code. However, the developer needs to understand the overall structure of the code so that he can create a clean implementation which utilizes reusable aspects of the code without damaging the code structure already in place.

### REFACTORING

Refactoring is a relatively new concept, this is the process by which the design of the code is actually improved, while leaving the functionality unchanged. This process can be done manually, using automated tools, or a combination of both. The main goal of refactoring is to reduce the cost of future maintenance by increasing the clarity and decreasing the size of the code.

The importance of Refactoring is being realized now in large projects, new programming methodologies such as extreme programming rely heavily on the power of refactoring for project development and code maintenance.

### A look to alternatives

There are numerous sources that state that code maintenance is time consuming and costly, most sources agree that maintenance will account for between 60% and 90% of the total IT budget. With such vast amounts of money spent in this area it seems appropriate that we find the metrics or pressure points that affect code maintenance.

## WHAT ARE THE ALTERNATIVES?

---

### CODE MAINTENANCE: TAKING A LESSON FROM COCOMO

**There are many packages for creating cost models for code projects. The COnstructive COst MOdel 2 (COCOMO) is one such tool that predicts the cost of a project from a number of key points.**

Over the last decade there has been a move towards the creation of generic reusable code (components) and object orientation. A new project will often reuse aspects of previous projects and, in one sense, it could be stated that almost no projects now start from a blank sheet of paper. There is some truth in the statement that all development projects are code maintenance, even in version 1 of the product. The same pressure points that COCOMO uses to predict costs for new projects must also be key factors in predicting the costs of code maintenance for the same project.

COCOMO is an open model that allows the equations that it uses for cost calculations to be studied. COCOMO is a rich source of information that allows the time intensive factors within a project to be detected. The problem of code maintenance can be broken down into these separate factors and informed strategies can be considered to reduce the overall code maintenance cost.

COCOMO calculates project length and cost using as its base calculation the number of lines of code needed for the project. This base number is modified by several exponential scale factors, such as precedence, flexibility, risk resolution, team cohesion and process maturity.

The secondary key factors – the Effort-Multiplier Cost Drivers – are grouped into four categories:

- Product.
- Platform.
- Personnel.
- Project factors.

Each category is a multiplier factor (nominally 1) that adjusts the overall cost of the project.

**Factors influencing project costs**

**Base calculation**

Estimate/Negotiate the cost of change                      The base measurement of length is a function of the number of lines of code.

**Effort-exponent cost drivers**

Issue of precedent	The uniqueness of the project being undertaken.
Development flexibility	The rigidity of the project goals.
Architecture/Risk resolution	A measure of the risk analysis performed on the project.
Team cohesion	A measure of the degree to which the development team integrates together.
Process maturity	The rating of the project as stipulated by the capability maturity model.

**Effort-multiplier cost drivers**

Product factors                      These factors describe the difficulty of the project under construction in terms of its requirements. These factors address issues such as:

- Code reliability.
- Size of the database.
- Product complexity.
- Required reusability.

Platform factors                      These factors address the platform on which the product will be mounted. This could be hardware, the operating system, even a web browser. These factors address such issues as:

- Execution speed.
- Platform support and fixes.
- Storage constraints.

Personnel factors                      These factors address the personnel and developers assigned to the product. These factors address such issues as:

- Analyst capability.
- Programmer capability.
- Applications experience.
- Platform experience.
- Language and tool experience.
- Personnel continuity.

Project factors                      These factors address the tools, site and deadlines that impact upon the project and such issues as:

- Code tools used.
- Multi-site development issues.
- Required development schedule.

**Learning from COCOMO**

COCOMO’s primary prediction factor is derived from the number of lines of code within the product. All COCOMO’s cost equations can be refined into a series of general statements about the IT project:

**Cost issues**

Inflexibility	Rigidly described projects with concrete goals are more expensive than projects that have flexible goals and can offer solutions to problems when they arise.
Unprecedented projects	Unique projects are much more expensive at the design stage than the maintenance stage due to the borderline research nature of the project. However unprecedented projects are also harder to maintain as the unprecedented nature of the project will still be an issue when the developer changes or extends the code.
A team weak on communication will slow	As the size of a development team increases, the individual speed of progress slows. Teams with weak communication will project development find themselves writing similar or redundant code due to the differing understandings of the code being produced.
Product factors such as large databases	<p>Project requirements greatly alter the cost of a project. The use of large databases requires good design and search complexity. High complexity and high reliability are all costly techniques to be implemented for timely archiving and retrieval of data.</p> <p>Developers have attempted to build scalable code that will not slow exponentially as the database size increases. The collection of data for testing and data cleansing are also costly.</p> <p>Increased complexity requires a greater amount of developer time devoted to understanding the problems at hand and a larger suite of tests is needed to capture areas where complexity can dull understanding.</p> <p>High reliability requires careful and extensive testing, especially when the code is modified. A clear view is needed of the code structure so that the implications of change are understood.</p>
Weak development tools	The use of refactoring tools and Integrated Development Environments to generate generic code greatly increases development speed. The use of new object-orientated languages has allowed encapsulation of code enabling it to be generated in parts, greatly reducing the complexity of large projects.
Platform tuning for increased execution performance	Similar to the issues regarding large databases.
Personnel factors such as inexperienced programmers untrained in the code used for development	The developers shape the project. The skill and capability of personnel impact on all other issues of the project. Capable personnel are critical to decreasing the cost.

## SOLUTIONS TO CODE MAINTENANCE

---

**COCOMO** has highlighted the IT issues that consume the most time and money during code maintenance. One technology that has the potential to massively reduce these issues is the Erudine Behaviour Engine.

### ERUDINE – A MAINTENANCE REVOLUTION

Erudine is not a programming language. It is a framework in which new projects can be built and old projects can be remodeled and maintained. With Erudine the distinction between building a new product and maintaining a product is blurred as it uses the same workflow for both building and maintaining the project.

The Erudine Behaviour Engine has been built with code maintenance in mind and several of the critical problems that dramatically increase the cost of traditional code maintenance are dealt with seamlessly by this unique technology. Erudine uses a patented Behaviour Engine that fulfils the role of an expert system, but is additionally supported by a suite of automatically generated tests.

Tests are of critical importance for validating the correct operation of the Behaviour System. When new behaviour is added to the system or when behaviour is changed the tests can be used to ensure that previous project functionality is not lost. The Erudine Behaviour Engine tests all aspects of its behaviour and enforces that old cannot be lost. It can be refined, but not lost.

To reiterate: there are four main areas of code maintenance. These are:

- Refinement.
- Change.
- Adding new functionality.
- Refactoring.

The Erudine Behaviour Engine directly addresses the difficulties within each of these areas.

### REFINEMENT AND ERUDINE

Refinement is handled very easily as refinement issues are addressed at the very heart of the Conclusion and Justification methodology used by the Erudine technology. (See [www.erudine.com](http://www.erudine.com) for more details)

#### Refinement example

A new situation (a new input) is presented to the system; The Erudine Behaviour Engine makes its conclusions based on its current behaviour set. The person maintaining the code decides whether the conclusions are correct and, if they are not, changes the conclusions.

This act of changing the conclusions is very important: it tells the Erudine Behaviour Engine that the current behaviour set needs expanding or refining, and it responds by working out which parts of the system will be affected by the new change. Counter-examples are presented to the code maintainer; this is behaviour that must be altered to make the developers changes consistent with the existing set.

The code maintainer needs to justify to the Erudine Behaviour Engine the differences between the current situation, and the counter behaviour. Only when these problems are resolved, can the new behaviour be accepted.

## CHANGE AND ERUDINE

Erudine is much less susceptible to problems arising from changing the functionality of the project, in fact there is no distinction between adding new functionality in a new project and altering an old existing project. It is inevitable that some requirements will change and so the Erudine Behaviour Engine uses a work flow that enables the system behaviour to be changed seamlessly.

The Erudine Behaviour Engine automates the difficult tasks of adding new functionality or changing behaviour. It uses a technique called Conclusion and Justification. This process is explained in the white paper 'Unlocking Tacit Knowledge by Conclusion and Justification'.

The Conclusion and Justification process used by the Erudine Behaviour Engine uses human readable data representation formats, and the process itself is straight-forwards. The consequence of this is that end users can be much more involved in the design process for the project, as the Erudine Behaviour Engine is much more accessible to them. This means that the clarity provided by the Erudine Behaviour Engine reduces mistakes in the initial requirements and narrows the communication gap between IT developers and the end users.

A change would be one where one or more of the previous behaviours is found to be wrong and requires alteration. A corrected conclusion is quite likely to impact onto other behaviour and all this is brought to the user's attention by the Erudine Behaviour Engine; all the behaviour can be modified by the user.

No changes can be implemented until all the consequences of the change are resolved. The system will not allow itself to get into a state where any of its internal tests fail; this is essential for maintaining the integrity of the behaviour base as the Erudine Behaviour Engine requires the 'mental model' of the code always to be consistent.

## ADDING NEW FUNCTIONALITY AND ERUDINE

Adding new behaviour in the Erudine Behaviour Engine is an almost identical process to refinement. New behaviour is added using the same language of Conclusion and Justification and the situation is stored as a test case and as an example of a situation where the behaviour is applicable.

When a new behaviour is added to using the Erudine Behaviour Engine the validity of the behaviour is tested against all previous tests. If the new behaviour alters the conclusion of an existing one then this difference needs to be resolved before the new behaviour is added. When all tests pass the test, then the behaviour can be added.

It is of critical importance that the Erudine Behaviour Engine checks the coherence of the entire behaviour set when new behaviour is added, this gives the user confidence that the new behaviour will not break some other obscure supposedly unrelated behaviour. The user is saved from the task of painstakingly checking the coherence of the code by hand.

## REFACTORING

Refactoring is the action of improving the design of existing code, while keeping the functionality the same. In the context of the Erudine Behaviour Engine, the code is the set of behaviour and supporting test cases that Erudine has generated. Optimisation can be defined in this case as reducing the behaviour in number while still providing the same functionality. The Erudine Behaviour Engine can perform simple optimisation checks on to remove redundant statements or behaviour.

## ERUDINE ADDRESSES THE LESSONS OF COCOMO

---

In this paper we have taken a detailed look at **COCOMO** and extracted from **COCOMO** a list of the pressure points of factors that affect the code cost of the project. The Erudine Behaviour Engine has been designed to eliminate the inherent weaknesses of a typical IT project.

### INFLEXIBILITY TO NATURAL ADAPTATION

The Erudine Behaviour Engine is a totally flexible environment. The behaviour is built up one situation at a time using Conclusion and Justification. As part of the capture of behaviour the Erudine Behaviour Engine will refine and change its behaviour and so offers seamless flexibility.

### UNPRECEDENTED PROJECTS

Unprecedented projects will always be more costly as the user will be working outside their normal comfort zones. However, the Erudine Behaviour Engine supports the user in tackling unusual situations.

Because it is an automated behaviour creation program it allows 'what if' scenarios to be tested quickly, and as new behaviour is added the scope of any changes are made explicit. This allows users to determine quickly if the changes that they are making are sensible; they can then quickly navigate through problems with no obvious solutions.

### WEAK COMMUNICATIONS

A team with weak communications impacts on a project in a number of ways. Although the Erudine Behaviour Engine is not a cure for poor communication, it can however help alleviate some of the symptoms of poor communication.

The Erudine Behaviour Engine allows the user to view the situations in which previous behaviour has been added to the Behaviour Engine. This provides a valuable glimpse into the thought processes of the developer who entered the behaviour, providing understanding and increasing project continuity between developers.

The Erudine Behaviour Engine will also present counter cases when a user is creating behaviour in contradiction to previous behaviour. This means that it will make the user aware of other changes made by other people to the project.

## PRODUCT FACTORS

Product factors such as large databases, high complexity and high reliability are all costly. The Erudine Behaviour Engine offers a number of benefits to such projects. It allows semantic searches of databases, this allows it to quickly refine a search and focus directly on relevant data. The Erudine Behaviour Engine also uses a revolutionary technology that gives execution of order zero with respect to the behaviour.

This means that there is only a small difference in execution time between processing 10 behaviour and processing 10,000 . This is essential when behaviour is being processed on a large database.

We can see from this that the Erudine Behaviour Engine embraces complexity. Through the refinement process previously described, complex situations will naturally be handled as the Erudine Behaviour Engine refines the system behaviour.

## WEAK DEVELOPMENT TOOLS

Erudine automates project construction using its Behaviour Engine. The user is free to reap the rewards of fast behaviour creation and behaviour validation processes as part of the Conclusion and Justification language.

## PLATFORM TUNING

Platform tuning for increased execution performance is costly but Erudine offers its order zero Behaviour Engine for the processing of business logic. A suite of refactoring tools for reshaping the behaviour is also available. This means that the Erudine Behaviour Engine is easily optimised to allow lightning fast execution of business logic.

## PERSONNEL FACTORS

While skilled development staff is mandatory for any IT project, the Erudine Behaviour Engine provides tools that assist personnel in project development. It supports the user's mental model of the project. Whether the user is adding or refining the behaviour set, Erudine will automatically track changes made and highlight the ramifications of the changes to the whole system; the user is not able to break any part of the system by inputting a new behaviour.

## THE FUTURE OF CODE MAINTENANCE

---

**The statistics on code maintenance make grim reading, with money being poured into the maintenance of hard to maintain code, with the developers in charge of maintenance often finding themselves poorly equipped to deal with this task. Code maintenance is a critical problem that currently drains 60-90% of the IT industry's total budget.**

As projects focus more on reusability, there is a blur in the distinction between maintenance and initial design. COCOMO is not only a cost model for predicting project build cost, but it can also be used as an indicator of code maintenance costs. This paper takes the practical view of using COCOMO to illustrate the pressure points that drive up code maintenance costs. This paper describes code maintenance by breaking it down into: Refinement; Changing your mind; Adding new functionality; and Refactoring.

The IT industry needs a fundamental change in large-scale system architecture and tools to develop these projects, to allow cheaper maintenance and better tools to support code maintenance.

This paper presents the Erudine Behaviour Engine as a revolutionary product for tackling the critical problems and costs of code maintenance. It describes how the Erudine Behaviour Engine with its automated test suite will greatly enhance developer's abilities to modify and maintain code. The Erudine Behaviour Engine greatly increases the efficiency of code maintenance, resulting in direct financial saving for any company maintaining a complex IT system.

## BIBLIOGRAPHY

---

- Eastwood, A. Firm Fires Shots at Legacy Systems. The Standish Group (1993).  
Erlikh, L. Leveraging Legacy System dollars for E-business. IEEE IT Pro (May/June 2000).  
Standish, T. An essay on code reuse. IEEE Transactions on Code Engineering (1984).  
Ulrich, W. The evolutionary growth of code engineering and the decade ahead. American Programmer (1990).  
Sommerville, I. Code Engineering, 6th edn. Addison-Wesley (2000).  
Rice, P. and Bamber, F. Unlocking Tacit Knowledge by Conclusion and Justification. Erudine (2004);

---

**WWW.ERUDINE.COM**

---

Tel: +44 (0) 8456 123 862

[technical@erudine.com](mailto:technical@erudine.com) / [www.erudine.com](http://www.erudine.com)

Design: [www.solutiongroup.co.uk](http://www.solutiongroup.co.uk)